23 VZOROVÝCH POSTUPŮ PRO RYCHLEJŠÍ VÝVOJ

Návrhové vzory v PHP

Marian Böhmer

Stručný přehledný výklad
Ukázky v UML a příklady z praxe
Alternativní řešení a postupy

computer press

Marian Böhmer

Návrhové vzory v PHP

Computer Press
Brno
2012

Návrhové vzory v PHP

Marian Böhmer

Překlad: Lukáš Krejčí
Obálka: Martin Sodomka
Odpovědný redaktor: Martin Herodek
Technický redaktor: Jiří Matoušek

Objednávky knih:
http://knihy.cpress.cz
www.albatrosmedia.cz
eshop@albatrosmedia.cz
bezplatná linka 800 555 513
ISBN 978-80-251-3338-5

Vydalo nakladatelství Computer Press v Brně roku 2012 ve společnosti Albatros Media a. s. se sídlem Na Pankráci 30, Praha 4. Číslo publikace 15 935.

1. vydání

ALBATROS MEDIA a.s.

Obsah

ČÁST II
TVOŘIVÉ VZORY

Úvodem

Návrhové vzory nabízejí postupy řešení často se vyskytujících problémů při návrhu softwaru. Často jsou považované za velmi komplikované, asociované s komplexními diagramy jazyka UML a obtížně pochopitelnými architekturami. Tato kniha vám ukáže, že návrhové vzory a jejich implementace v jazyce PHP nejsou až tak komplikované, jak se mohou někomu jevit po prvním přečtení jejich definice. Na praktických příkladech se seznámíte s různými návrhovými vzory, které vám pomohou při každodenní práci a kvůli nimž už nebudete muset organizovat velká pracovní střetnutí nebo vytvářet vícestranné diagramy. Pomocí návodů v této knize se pro vás návrhové vzory stanou nástrojem, který vám ulehčí a obohatí chvíle při vývoji aplikací v jazyce PHP.

Jazyk PHP 5 přinesl množství novinek, například kompletně přepracovanou podporu pro XML nebo jednodušší použití webových služeb přes protokol SOAP, který je automaticky podporovaný v jazyce PHP. Přístup k různým typům databází lze nyní díky PHP 5.1 a PDO provádět přes jednotné rozhraní.

Naproti tomu tato aplikační rozhraní sama o sobě zatím neumožňovala realizovat profesionální aplikace. O to se postaral až Zend Engine 2 s kompletně přepracovaným objektovým modelem, který nabízí viditelnost pro atributy a metody, rozhraní, výjimky, ale i abstraktní a finální třídy. To vám, jakožto vývojářům v jazyce PHP, umožňuje navrhovat softwarové architektury, které nebudou v ničem zaostávat za architekturami vývojářů v jazyce Java. Tato kniha vám ukáže, jak můžete při návrhu softwaru využít prvky jazyka PHP 5.3 k tomu, aby váš software splňoval moderní standardy a bylo jej možné bez problémů rozšířit v případě, že se změní požadavky na aplikaci.

Dále se seznámíte s pravidly, která byste měli dodržovat při návrhu aplikace. Kromě toho vám návrhové vzory nabízejí jazyk, pomocí něhož můžete řešit problémy s ostatními kolegy vývojáři, aniž abyste museli každý detail podrobně vysvětlovat.

Struktura knihy

Celá kniha je rozdělená do pěti částí a lze ji číst dvěma způsoby. Buď ji budete číst postupně, od začátku do konce, a přitom se naučíte krok za krokem jednotlivé návrhové vzory, nebo ji použijete jako katalog, v němž si vyhledáte přesně ten návrhový vzor, jehož implementací se chce zabývat. Každý návrhový vzor představuje jednu kapitolu. V některých kapitolách se setkáte s odkazy na jiné návrhové vzory – můžete je však použít i nezávisle na sobě. Jiné kapitoly naopak nemusí být rozepsané dopodrobna, a to zvláště v případě, že popisovaný návrhový vzor lze nahradit jiným návrhovým vzorem, který řeší stejný problém.

V úvodní části (kapitola 1 – Základní pravidla při návrhu softwaru) si na příkladu vyzkoušíte, jakým chybám byste se měli vyhnout při návrhu architektury softwaru. Vyvodí se z nich všeobecně platná pravidla, jež se stanou základem návrhových vzorů probíraných v této knize.

Druhá část (kapitoly 2–6) pojednává o návrhových vzorech týkajících se tvorby objektů.

V třetí části knihy (kapitoly 7–13) se představí návrhové vzory, které se zabývají kompozicí různých objektů. V této části se navíc seznámíte se strategiemi, které umožňují rozšiřitelnost daných kompozic.

Čtvrtá část (kapitoly 14–24) se zabývá poslední skupinou klasických návrhových vzorů. Tyto vzory řeší problémy, které se mohou často vyskytnout při interakci různých objektů.

V poslední, páté části (Příloha) najdete krátký úvod do jazyka UML, díky němuž nebudete mít problém pochopit diagramy jazyka UML v jednotlivých kapitolách. Kromě toho v této části najdete i techniky nazývané vkládání závislostí (angl. Depencency Injection), které staví na filozofii převráceného řízení (angl. Inversion of Control – IoC), a také se naučíte vytvářet aplikační rozhraní s Fluent Interfaces.

Ve všech výpisech v knize budou vynechané počáteční a koncové značky (`<?php` a `?>`) a bude uvedený jen kód jazyka PHP. Pro označení názvů jednotlivých tříd, metod a atributů budeme používat anglické pojmy, což je způsob, který doporučuji i pro vaše projekty, protože tyto pojmy se lépe hodí k nativním názvům tříd a funkcí jazyka PHP.

Pro lepší přehlednost nebudou uváděné komentáře. U reálných projektů byste však měli myslet alespoň na popis aplikačního rozhraní pomocí nástroje PHPDoc (*http://www.phpdoc.org*).

Všechny příklady jsou napsané pro jazyk PHP ve verzi 5.3. Chcete-li je použít se starší verzí jazyka PHP, musíte je patřičně upravit – vzdát se používání jmenných prostorů a odstranit z výpisů klíčová slova `namespace` a `use`.

Komu je kniha určena

Tato kniha je určená vývojářům, kteří už mají zkušenosti s programováním v jazyce PHP. V ideálním případě se už při realizaci projektu střetli s objektově orientovaným programováním (OOP) v jazyce PHP 4 nebo PHP 5, avšak žádné hluboké znalosti OOP v jazyce PHP 5 nejsou podmínkou.

Pro vysvětlení a pochopení jednotlivých návrhových vzorů probíraných v této knize nejsou vyžadované žádné hluboké znalosti jazyka UML (Unified Modeling Language). Kromě krátkého úvodu do tohoto jazyka, který najdete v příloze, nabízí tato kniha diagramy, jejichž obsah pro vás bude srozumitelný i se základy jazyka UML.

Kniha Návrhové vzory v PHP je napsaná pro programátory, kteří chtějí při vytváření profesionálních architektur využívat vlastnosti OOP. Jejím těžištěm je přitom architektura softwaru. Věci, které v knize nebudou probírané, se týkají oblasti nízkoúrovňové logiky, jako je například přístup k souborům, analyzování dokumentů XML nebo přístup k údajům v databázi pomocí jazyka SQL.

Máte-li málo zkušeností s jazykem PHP, avšak pojem návrhové vzory vám je známý z jiných programovacích jazyků, můžete se pomocí této knihy seznámit s implementací různých vzorů, jejichž realizace uvedeným způsobem je možná jen v jazyce PHP.

Zpětná vazba od čtenářů

Nakladatelství a vydavatelství Computer Press, které pro vás tuto knihu připravilo, stojí o zpětnou vazbu a bude na vaše podněty a dotazy reagovat. Můžete se obrátit na následující adresy:

redakce PC literatury
Computer Press
Spielberk Office Centre
Holandská 3
639 00 Brno

nebo

sefredaktor.pc@cpress.cz

Computer Press neposkytuje rady ani jakýkoli servis pro aplikace třetích stran. Pokud budete mít dotaz k programu, obraťte se prosím na jeho tvůrce.

Errata

Přestože jsme udělali maximum pro to, abychom zajistili přesnost a správnost obsahu, chybám se úplně vyhnout nedá. Pokud v některé z našich knih najdete chybu, ať už chybu v textu nebo v kódu, budeme rádi, pokud nám ji nahlásíte. Ostatní uživatele tak můžete ušetřit frustrace a pomoci nám zlepšit následující vydání této knihy.

Veškerá existující errata zobrazíte na adrese *http://knihy.cpress.cz/K1886* po klepnutí na odkaz Soubory ke stažení.

ČÁST I

Úvod do návrhu softwaru

KAPITOLA 1

Základní pravidla při návrhu softwaru

Máloco podléhá tak častým změnám jako právě software. Ve většině projektů se na něj nakládají stále nové požadavky a stejně rychle odpadají požadavky dosud implementované – nebo v lepším případě jsou často pozměněné. Jako softwarový vývojář musíte jednotlivé komponenty své aplikace navrhnout takovým způsobem, abyste mohli relativně rychle a flexibilně reagovat na měnící se požadavky. V této kapitole se dozvíte, jak lze vyvíjet flexibilní aplikace, a také to, že je důležité zamyslet se nad návrhem aplikace ještě před vlastní implementací. Tím se vyhnete situaci, kdy náhle stojíte před problémem, kdy musíte přepsat velké části aplikace, aby bylo možné reagovat na změněné požadavky. Jestliže tomu nezabráníte, může každý změněný požadavek na software znamenat zvýšené náklady, čímž se prodlouží čas nutný k dokončení projektu nebo se do softwaru nestihnou zapracovat všechny požadavky.

Na příkladu v této kapitole si osvojíte základy návrhu softwaru a naučíte se, jak sestavit aplikaci skládající se z více tříd, jež splňují požadavky na rozšiřitelnost a flexibilitu.

Na konci kapitoly se dozvíte, jak lze co nejrychleji najít vhodný návrhový vzor pro řešení budoucích problémů. Vyzbrojeni těmito vědomostmi můžete použít následující kapitoly jako encyklopedii, když narazíte v jednom ze svých projektů na problémy, které budete chtít vyřešit pomocí návrhových vzorů.

Protože hlavním cílem této knihy je blíže vás seznámit se správnou architekturou aplikací, výklad nebude zacházet až do detailů a nebude se věnovat perzistentnímu ukládání údajů nebo grafickému uživatelskému rozhraní. Místo toho budeme klást důraz na rozhraní, která vám dané třídy nabízejí, a na to, jak jednotlivé třídy mezi sebou komunikují.

Pravidla softwarového návrhu

K objasnění základních pravidel softwarového návrhu využijete následující příklad. Představte si, že byste měli implementovat aplikaci, pomocí níž bude knihovna spravovat své publikace. Na konci této kapitoly budete mít za sebou vývoj funkční aplikace, s níž bude možné tyto publikace půjčovat a každé takovéto půjčení přesně protokolovat. V reálné knihovně bude správce chtít jistě ukládat o každé publikaci více informací a půjčování nebude nefungovat přesně tak, jako to bude uvedené zde, ale jako příklad probírané problematiky to stačí.

Pokud jste se už nějakým způsobem zabývali objektově orientovaným programováním, jsou vám pojmy *třída* (angl. *Class*), *objekt* (angl. *Object*) a možná i rozhraní (angl. *Interface*) docela dobře známé. Koncepce tříd představuje pokus přenést věci z reálného světa, s nimiž má aplikace něco společné, do světa programování.

Každý typ publikace má samozřejmě jiné vlastnosti, ale určitým způsobem se všechny vzájemně podobají, z čehož lze odvodit následující rozhraní:

```
namespace cz\k1886\example;

interface Publication {
    public function open();
    public function close();
    public function setPageNumber($page);
    public function getPageNumber();
}
```

Reprezentace libovolné knihy by potom mohla vypadat následovně:

```
namespace cz\k1886\example;

class Book implements Publication {
    protected $category;
    protected $pageCount;
    protected $pageNumber = 0;
    protected $closed = true;
```

```
    public function __construct($category, $pageCount) {
        $this->category = $category;
        $this->pageCount = $pageCount;
    }
    public function open() {
        $this->closed = false;
    }
    public function setPageNumber($page) {
        if ($this->closed !== false || $this->pageCount < $page) {
            return false;
        }
        $this->pageNumber = $page;
        return true;
    }
    public function getPageNumber()
    {
        return $this->pageNumber;
    }
    public function close() {
        $this->setPageNumber(0);
        $this->closed = true;
    }
    public function __destruct() {
        if (!$this->closed) {
            $this->close();
        }
    }
    // další metody pro čtení atributů
}
```

Reprezentace knihy má čtyři atributy: kategorii, do níž náleží, počet stran, konkrétní stranu, na níž je právě otevřená, a příznak udávající, zda je otevřená nebo zavřená.

Zapouzdření údajů

Třídy a objekty vám umožňují omezit přístup k údajům, čehož byste měli využívat. Díky tomu totiž můžete změnit vnitřní strukturu třídy, aniž by se to dotklo jiných tříd nebo metod, které dotyčnou třídu používají. Takové změny mohou být kromě jiného způsobené i následujícími důvody:

- Změnily se požadavky na aplikaci, přičemž je nutné změnit algoritmy. Pokud se změny provedou v rámci jedné třídy, neovlivní to žádnou z tříd, jež upravenou třídu volají.
- Implementované algoritmy jsou velmi pomalé, protože aplikaci používá stále větší počet uživatelů. Tyto algoritmy můžete přepsat, povedou-li ke stejnému výsledku.
- Je zapotřebí změnit úložiště údajů. K tomu může dojít například v situaci, když zjistíte, že aktuální úložiště je vzhledem k objemu údajů nevhodné, a je tedy nutné použít místo souboru databázi.

Pro objasnění znovu využijeme příklad knihovny. Vzhledem k tomu, že půjčování publikací není zadarmo, musí být někde uložená denní sazba pro jednu publikaci. Je-li tato hodnota zpočátku stále stejná, může vás napadnout použít globální proměnnou nebo nějaký podobný způsob konfigurace:

```
$dailyRate = 10;
```

Co se ale stane, přijde-li požadavek, aby se denní sazba pro dětské knihy lišila od sazby pro knihy vědecké nebo aby celková částka závisela na délce vypůjčení. Tím by velmi vzrostl počet konfiguračních možností a museli byste v příslušných místech své aplikace implementovat logiku, pomocí níž byste načítali správnou konfiguraci.

Z těchto důvodů je lepší zapouzdřit přístup k denní sazbě hned od začátku. Protože se předpokládá, že výška této sumy se bude lišit v závislosti na vypůjčené publikaci, představuje rozhraní `Publication` dobré místo pro tuto logiku. Toto rozhraní totiž implementují všechny publikace (knihy, časopisy apod.). Rozšiřte tedy toto rozhraní o jednu metodu a tu pak implementujete v příslušných třídách. Příklad metody pro třídu `Book` by mohl vypadat následovně:

```
namespace cz\k1886\example;

class Book implements Publication {
    // ... atributy a metody

    public function getDailyRate() {
        return 10;
    }
}
```

Má-li být pro jiný typ knihy účtovaná jiná denní sazba, pak stačí jen přepsat tuto metodu a nový požadavek máte implementovaný. Tím jste se seznámili s prvním pravidlem návrhu softwaru u objektově orientované architektury:

PRAVIDLO

Přístup k údajům vždy v rámci třídy zapouzdřete a poskytněte metody, pomocí nichž lze dané údaje získat.

Stejně jednoduchá je implementace požadavku, aby denní sazba byla nižší v případě vypůjčení knihy na déle než dva týdny. V tomto případě stačí také změnit jen jednu metodu:

```
namespace cz\k1886\example;

class Book implements Publication {
    // ... atributy a metody

    public function getDailyRate($days = 1) {
        if ($days >= 14) {
            return 7.50;
        }
        return 10;
    }
}
```

S tím také samozřejmě souvisí úprava tříd, jež volají metodu `getDailyRate()`, protože tato metoda musí od nynějška předávat počet dní. Této dodatečné změně se můžete vyhnout, pokud jste se při návrhu rozhraní pokusili stanovit, jaké údaje by byly eventuálně potřebné při výpočtu denní sazby. Nikdy však nebude možné zohlednit všechny budoucí požadavky hned při první implementaci rozhraní. V takovém případě byste museli metodě předávat všechny dostupné údaje, čímž byste rozbili zapouzdření údajů. Pokuste se najít zlatou střední cestu a metodu navrhnout tak, aby byla v budoucnosti jednoduše rozšiřitelná – jak jsme si to ukázali v tomto příkladu. Tím jste se naučili už druhé pravidlo objektově orientovaného návrhu softwaru:

PRAVIDLO

Svá rozhraní navrhujte tak, aby je bylo možné později rozšířit.

Na začátku návrhu softwaru provedete vždy stejné kroky: musíte se zamyslet nad spravovanými údaji a nad tím, jak lze tyto údaje zahrnout do entit. Z toho pak vyplynou jednotlivé třídy a metody.

Aktéři

V příkladu knihovny se nacházejí následující aktéři, kteří musí být částí vaší aplikace:

- jednotlivé knihy, které jsou částí knihovny a lze je půjčovat,
- členové knihovny, kteří si knihy půjčují,
- samotná knihovna, která spravuje jednotlivé publikace a stará se o půjčování.

Z těchto představitelů můžete hned odvodit potřebné třídy a rozhraní. Pro knihy jste to už provedli, zůstávají už jen členové a knihovna. Implementujte proto třídu `Member`, která bude reprezentovat jednoho člena knihovny:

```
namespace cz\k1886\example;

class Member {
    protected $id;
    protected $name;

    public function __construct($id, $name) {
        $this->id = $id;
        $this->name = $name;
    }

    public function getId() {
        return $this->id;
    }

    public function getName() {
        return $this->name;
    }
}
```

Daný člen se skládá z jedinečného identifikátoru, jímž se v systému identifikuje, a ze svého jména, což pro jednoduchou demonstraci úplně stačí. Informace, které jsou přiřazené členovi, se předají prostřednictvím konstruktoru a uloží se do atributů objektu. Přístup k těmto údajům je možný pomocí metod `getId()` a `getName()`.

Po vytvoření tříd pro členy a knihy nastal čas věnovat se samotné knihovně, kterou reprezentuje třída `Library`:

```
namespace cz\k1886\example;

class Library {
    protected $library = array();

    public function addToLibrary($id, Publication $p) {
        $this->library[$id] = $p;
    }

    public function rentPublication(Publication $p, Member $m) {
    }

    public function returnPublication(Publication $p) {
    }
}
```

Knihovna má jeden atribut, do něhož se ukládají všechny publikace určené k vypůjčení, a také tři metody:

- Metoda `addToLibrary()` se používá k přiřazení nové publikace do knihovny. Tato publikace se jednoduše uloží do pole `$library`.
- Pomocí metody `rentPublication()` si může člen knihovny vypůjčit konkrétní publikaci. Tuto metodu jste zatím neimplementovali, neboť nejdříve je nutné definovat, v jaké formě se budou údaje ukládat.
- Pro vrácení publikace se používá metoda `returnPublication()`. V tomto případě není nutné předat objekt člena, který publikaci vrací. Koneckonců knihovna přece musí vědět, kdo měl danou publikaci vypůjčenou.

V následujících krocích budete deklarovat metody naplněné skutečnou logikou. Metody `rentPublication()` a `returnPublication()` mají vždy něco společné s vypůjčením publikace. Buď proces vypůjčení začal, nebo byl vrácením publikace zpět do knihovny ukončený.

Tohoto procesu se vždy účastní dva aktéři: publikace, která má být vypůjčena, a osoba, která si ji vypůjčila. Dále jsou důležité časové údaje, kdy byla publikace vypůjčená a také kdy byla vrácená, aby tak bylo možné vypočítat cenu za vypůjčení. Ve skutečné knihovně to většinou funguje jinak, ale jako příklad postačí i tato varianta. Podle pravidel zapouzdřování, která jste se právě naučili, se pokuste dostat tyto informace do jedné třídy.

Implementace procesů půjčování

Nová třída `RentalAction` musí obsahovat následující informace:

- osoba, která si publikaci půjčuje,
- publikace, která se půjčuje,
- datum, kdy byla publikace vypůjčena,
- datum, kdy byla publikace vrácena a proces ukončen.

Implementace této třídy může vypadat následovně:

```
namespace cz\k1886\example;

class RentalAction {
    protected $publication;
    protected $member;
    protected $rentDate;
    protected $returnDate = null;

    function __construct(Publication $p, Member $m, $date = null) {
        $this->publication = $p;
        $this->member = $m;

        // v případě neuvedení data použít aktuální
        if (null === $date) {
            $date = date('Y-m-d H:i:s');
        }
        $this->rentDate = $date;
    }

    public function getPublication() {
        return $this->publication;
    }

    public function getMember() {
        return $this->member;
    }

    public function getRentDate() {
        return $this->rentDate;
    }

    public function getReturnDate() {
        return $this->returnDate;
```

```
    }
}
```

Konstruktoru třídy musíte při vytvoření nového procesu předat jako parametry publikaci, která má být vypůjčena, a dále osobu, která si ji půjčuje. Volitelně můžete uvést datum a čas, kdy byla daná publikace vyzvednuta. V případě neuvedení data se použije aktuální datum. Tyto hodnoty se uloží v příslušných atributech třídy. Dále se ve třídě nacházejí čtyři metody pro čtení atributů, které vám umožňují přistupovat k atributům třídy. Pokud si nyní přijde nějaký člen vaší knihovny vypůjčit určitou publikaci, můžete vytvořit odpovídající proces vypůjčení takto:

```
use cz\k1886\example\Book;
use cz\k1886\example\Member;
use cz\k1886\example\RentalAction;

$book = new Book('PC', 100);
$mabo = new Member(1, 'Marian Böhmer');
$rentalAction = new RentalAction(
    $book, $mabo, '2011-08-22 16:00:00'
);
```

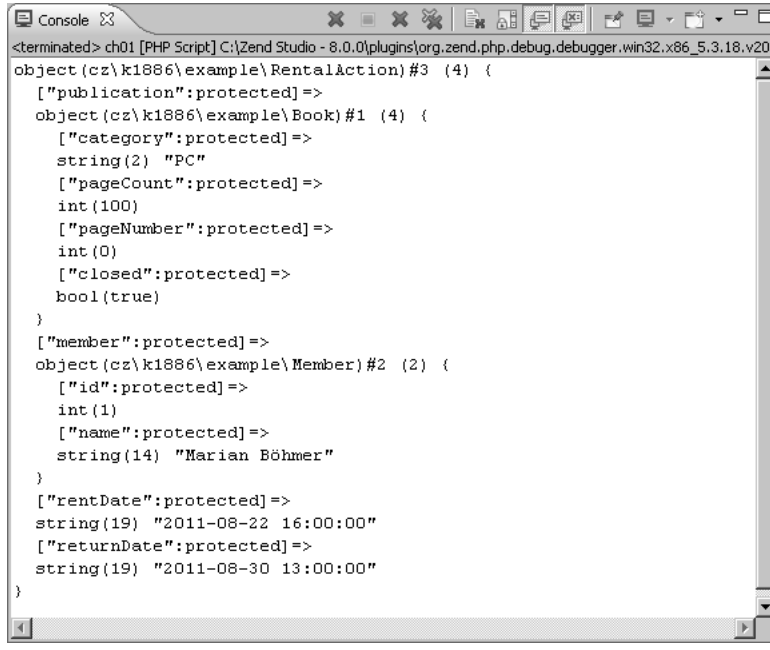
V této chvíli je sice možné knihu vypůjčit, avšak zatím není možné zaznamenat, že už byla i vrácena. Pro tento účel jste si v této třídě už předem rezervovali atribut `$returnDate`. K jeho nastavení je nutné vložit do třídy `RentalAction` následující metodu:

```
namespace cz\k1886\example;

class RentalAction {
    // ... atributy a metody třídy

    public function markPublicationReturned($date = null) {
        // v případě neuvedení data použít aktuální
        if (null === $date) {
            $date = date('Y-m-d H:i:s');
        }
        $this->returnDate = $date;
    }
}
```

Bude-li publikace přinesena zpět do knihovny, pak pro uložení této informace do objektu stačí zavolat příslušnou metodu:

```
$rentalAction->markPublicationReturned('2011-08-30 13:00:00');
```

Výsledek této operace je zobrazený na obrázku 1.1.



Obrázek 1.1: Proces vypůjčení knihy z knihovny

Pokud si chcete ověřit, zda byla kniha vrácena a tím daný proces ukončen, můžete k tomu využít hodnotu, kterou vrací metoda `getReturnDate()`:

```
if (null !== $rentalAction->getReturnDate()) {
    print 'Publikace byla vrácená';
}
```

Tento kód se však neimplementuje jednoduše, a proto je vhodné zapouzdřit jej do další metody:

```
namespace cz\k1886\example;

class RentalAction {
    // ... atributy a metody třídy

    public function isReturned() {
        return null !== $this->returnDate;
    }
}
```

Tuto metodu můžete nyní využít ke zjištění, zda už byla publikace vrácena:

```
if  ($rentalAction->isReturned()) {
    print 'Publikace byla vrácená';
}
```

Tím jste se naučili další pravidlo vývoje softwaru:

PRAVIDLO

V metodách tříd nezapouzdřujte jen údaje, ale také algoritmy, díky čemuž budou komplexní operace implementované centrálně na jednom místě.

Zavedení metody `isReturned()` vám v budoucnosti umožní na základě podmínky určit, zda už byl proces vypůjčení ukončen a kniha vrácena zpět do knihovny.

Pomocí nové třídy `RentalAction` je možné implementovat chybějící metody ve třídě `Library`. Nejdříve je nutné vložit do uvedené třídy nový atribut `$rentalActions`, v němž budou v podobě objektů uložené jednotlivé procesy vypůjčení. Nejjednodušším datovým typem použitelným k tomuto účelu je pole.

Následující implementace metody pro vypůjčení publikace musí splňovat následující kritéria:

- otestovat, zda je požadovaná publikace součástí knihovny,
- otestovat, zda je požadovaná publikace aktuálně vypůjčená (v takovém případě ji nelze znovu vypůjčit),
- pomocí nové instance třídy `RentalAction` vytvořit nový proces vypůjčení.

Implementace metody `rentPublication()` může vypadat následovně:

```
namespace cz\k1886\example;

class Library {
    protected $rentalActions = array();
    // ... atributy a metody třídy

    public function rentPublication(Publication $p, Member $m) {
        $publicationId = array_search($p, $this->library);
        if (false === $publicationId) {
            throw new UnknownPublicationException();
        }
        if (!$this->isPublicationAvailable($p)) {
            throw new PublicationNotAvailableException();
```

```
            }
            $rentalAction = new RentalAction($p, $m);
            $this->rentalActions[] = $rentalAction;

            return $rentalAction;
        }
    }
```

Pomocí funkce `array_search()` jazyka PHP můžete zjistit, zda se publikace nachází v nabídce knihovny. Vrátí-li hodnotu `false`, pak signalizuje chybu vyvoláním výjimky. Test, zda je daná publikace právě vypůjčená, je trochu komplikovanější, a proto je jeho logika přesunuta do další metody. Díky tomu mohou tuto logiku použít i jiné metody třídy. Tímto způsobem opět zapouzdřujete určitý algoritmus do jedné metody. Konkrétní implementace této metody bude probrána později. V případě, že publikace není vypůjčená, vytvoří se nová instance třídy `RentalAction`, které předáte půjčovanou publikaci a člena, který si ji půjčuje. Tuto instanci potom uložíte do k tomu určeného atributu, kde jsou uložené i všechny ostatní procesy vypůjčení. Metoda `rentPublication()` vrací objekt `RentalAction`. Vrácení tohoto objektu sice není v současnosti pro knihovnu důležité, protože neobsahuje žádné další informace než ty, které už obdržela, avšak v budoucnosti může objekt `RentalAction` obsahovat i další informace, například číslo objednávky, které budete chtít poskytnout aplikaci.

Ještě před tím, než začnete ve vypůjčené publikaci listovat, je nutné podívat se na implementaci pomocné metody `isPublicationAvailable()`. Pro zjištění, zda je daná publikace právě vypůjčená, stačí jen otestovat, zda atribut `$rentalActions` obsahuje proces vypůjčení pro hledanou publikaci, který ještě nebyl ukončen. K tomuto účelu vám dobře poslouží metody pro čtení atributů třídy `RentalAction`:

```
public function isPublicationAvailable(Publication $p) {
    foreach ($this->rentalActions as $rentalAction) {
        if  ($rentalAction->getPublication() !== $p) {
            continue;
        }
        if  ($rentalAction->isReturned()) {
            continue;
        }
        return false;
    }
    return true;
}
```

Ve funkci `isPublicationAvailable()` probíhá iterace přes všechny prvky v poli `$rentalActions` (všechny procesy vypůjčení). Pokud aktuální proces nepatří hledané publikaci nebo už je ukončený, pokračuje se s další iterací. Pokud daný proces náleží k hledané publikaci a ještě není ukončen, nemůže být daná publikace znovu vypůjčena, což se signalizuje vrácením hodnoty `false`. Neexistuje-li pro hledanou publikaci žádný proces, který ještě nebyl ukončen, vrátí se hodnota `true`.

V tomto okamžiku lze třídu `Library` použít k půjčování publikací a navíc je zaručené, že jedna publikace může být vypůjčená jen jednou, jak to ukazuje následující příklad:

```
use cz\k1886\example\Library;
use cz\k1886\example\Book;
use cz\k1886\example\Member;

$library = new Library();
$book = new Book('PC', 100);
$mabo = new Member(1, 'Marian Böhmer');
$luigi = new Member(2, 'Luigi Valentino');

$library->addToLibrary('pc1', $book);
$library->rentPublication($book, $mabo);
$library->rentPublication($book, $luigi);
```

Po jeho provedení reaguje knihovna následující výjimkou:

```
Fatal error:  Uncaught exception 'cz\k1886\example\
PublicationNotAvailableException' in ch01\Library.php:24
```

Tuto výjimku můžete ve frontendu své aplikace změnit na chybové hlášení.

Aby bylo možné i vrácení vypůjčených publikací, musíte dále implementovat metodu `returnPublication()`. Tato metoda musí najít aktuální proces vypůjčení, který náleží k dané publikaci, a označit jej jako ukončený. Kód této metody se podobá tomu v metodě `isPublicationAvailable()`. I v tomto případě procházíme přes všechny procesy vypůjčení, dokud nenajdeme proces pro hledanou publikaci, který ještě nebyl ukončen. Tento proces potom označíme pomocí metody `markPublicationReturned()` jako ukončený. Kompletní implementace vypadá takto:

```
namespace cz\k1886\example;

class Library {
```

```
    // ... atributy a metody třídy
    public function returnPublication(Publication $p) {
        foreach ($this->rentalActions as $rentalAction) {
            if ($rentalAction->getPublication() !== $p) {
                continue;
            }
            if ($rentalAction->isReturned()) {
                continue;
            }
            $rentalAction->markPublicationReturned();
            return true;
        }
        return false;
    }
}
```

V tomto okamžiku mohou být vypůjčené publikace znovu vráceny a připraveny na své další vypůjčení:

```
use cz\k1886\example\Library;
use cz\k1886\example\Book;
use cz\k1886\example\Member;

$library = new Library();
$book = new Book('PC', 100);
$mabo = new Member(1, 'Marian Böhmer');
$luigi = new Member(2, 'Luigi Valentino');

$library->addToLibrary('pc1', $book);
$library->rentPublication($book, $mabo);
$library->returnPublication($book);
$library->rentPublication($book, $luigi);
```

Při provedení tohoto kódu nedošlo k vyvolání žádné výjimky. Místo toho se vytvořily dva procesy vypůjčení, z nichž jeden byl i ukončený.

U této implementace vás možná napadne, proč není proces po jeho ukončení z pole vymazán? Jednoduše proto, že byste v takovém případě ztratili historii procesů vypůjčení. Představte si, že by chtěl správce knihovny zjistit, které publikace se nejvíce půjčují nebo kdo je nejaktivnějším členem knihovny. Pomocí procesů uložených v poli `$rentalActions` lze tyto údaje snadno poskytnout.

Je nutné si uvědomit, že v tomto příkladu nešlo o uchovávání údajů ani o jeho výkonnost, ale o vytvoření architektury aplikace. Místo pole s objekty typu `Ren-`

talAction byste v reálné aplikaci použili pro jejich uložení databázovou tabulku. Databázové systémy už nabízejí nástroje, jak lokalizovat záznam k aktuálnímu procesu vypůjčení konkrétní publikace.

Ladění aplikace

V této chvíli máte za sebou základy aplikace a jste schopni půjčovat publikace a přijímat je zpět. V této podkapitole si na příkladu části aplikace ukážeme, jaká další pravidla návrhu softwaru byste měli ve svých aplikacích zohlednit.

Třebaže funkce pro *ladění* (angl. debugging) aplikací v jazyce PHP lze zlepšit pomocí externích nástrojů, skoro každý vývojář v jazyce PHP využívá k výpisu nezbytných informací při tomto kroku příkaz `print` nebo `echo`. Tyto příkazy se během vývoje jednoduše vloží do zdrojového kódu a před tím, než se aplikace nasadí do provozu, se zase odstraní. V rámci této části budete takovýto ladicí kód postupně, kroku za krokem vylepšovat, při čemž se seznámíte s různými pravidly, která byste měli zohlednit i v dalších softwarových projektech.

Při dalším vývoji softwaru knihovny můžete do zdrojového kódu následujícím způsobem vložit hlášení používané při ladění:

```
namespace cz\k1886\example;
use cz\k1886\example\RentalAction;

class Library {
    protected $library = array();
    protected $rentalActions = array();

    public function addToLibrary($id, Publication $p) {
        $this->library[$id] = $p;
        print 'Nová publikace v knihovně: '
              . $p->getCategory() . "\n";
    }

    public function rentPublication(Publication $p, Member $m) {
        $publicationId = array_search($p, $this->library);
        if (false === $publicationId) {
            throw new UnknownPublicationException();
        }
        if (!$this->isPublicationAvailable($p)) {
            throw new PublicationNotAvailableException();
        }
```

```
            $rentalAction = new RentalAction($p, $m);
            $this->rentalActions[] = $rentalAction;

            print $m->getName() . ' si vypůjčil publikaci: '
                  . $p->getCategory() . "\n";
            return $rentalAction;
        }

        public function returnPublication(Publication $p) {
            foreach ($this->rentalActions as $rentalAction) {
                if ($rentalAction->getPublication() !== $p) {
                    continue;
                }
                if ($rentalAction->isReturned()) {
                    continue;
                }
                $rentalAction->markPublicationReturned();
                print $rentalAction->getMember()->getName()
                      . ' vrátil publikaci: '
                      . $p->getCategory() . "\n";
                return true;
            }
            return false;
        }
        // ... ostatní metody
    }
```

Pokud nyní znovu zavoláte příkazy z předchozího testovacího skriptu s pozměněnou třídou `Library`, můžete přesně sledovat, kdy se která metoda volá:

```
Nová publikace v knihovně: PC
Marian Böhmer si vypůjčil publikaci: PC
Marian Böhmer vrátil publikaci: PC
Luigi Valentino si vypůjčil publikaci: PC
```

V závislosti na tom, jakou část aplikace chcete ladit, se bude měnit i množství ladicích informací. Před přesunem aplikace na produkční server jednoduše tyto řádky kódu vymažete.

Co se ale stane, pokud objevíte problém, který se vyskytuje jen v produkčním prostředí, protože souvisí například s množstvím paralelních dotazů na aplikaci? Tato hlášení nemůžete jednoduše zobrazit koncovému uživateli. V takovém případě používá většina vývojářů protokolovací soubor, do něhož přesměrují veškerá hlášení. Často je tedy ladicí kód rozšířený o příkazy `if/else` nebo `switch`,

pomocí nichž lze přepínat mezi laděním v produkčním nebo vývojovém prostředí. V případě knihovny by to vypadalo následovně:

```
namespace cz\k1886\example;

class Library {
    protected $library = array();
    protected $rentalActions = array();

    public function addToLibrary($id, Publication $p) {
        $this->library[$id] = $p;
        switch(DEBUG_MODE) {
            case 'echo':
                print 'Nová publikace v knihovně: '
                      . $p->getCategory() . "\n";
                break;
            case 'log':
                error_log('Nová publikace v knihovně: '
                    . $p->getCategory() . "\n",
                    3, './library.log');
                break;
        }
    }

    public function rentPublication(Publication $p, Member $m) {
        $publicationId = array_search($p, $this->library);
        if (false === $publicationId) {
            throw new UnknownPublicationException();
        }
        if (!$this->isPublicationAvailable($p)) {
            throw new PublicationNotAvailableException();
        }
        $rentalAction = new RentalAction($p, $m);
        $this->rentalActions[] = $rentalAction;
        switch(DEBUG_MODE) {
            case 'echo':
                print $m->getName() . ' si vypůjčil publikaci: '
                      . $p->getCategory() . "\n";
                break;
            case 'log':
                error_log($m->getName() . ' si vypůjčil publikaci: '
                    . $p->getCategory() . "\n",
                    3, './library.log');
```

```
                break;
            }
            return $rentalAction;
        }
        // ... ostatní metody třídy
    }
```

Pomocí konstanty `DEBUG_MODE` můžete přepínat mezi jednoduchými hlášeními vypisovanými na obrazovku a zapisovanými do protokolovacího souboru funkcí `error_log()`. K tomu je nutné přidat do skriptu následující řádek, umožňující centrální řízení způsobu ladění:

```
define('DEBUG_MODE', 'log');
```

Co však na prvý pohled vypadá jako komfortní řešení, je v konečném důsledku jen velká slabina aplikace. Na každém místě, kde chcete vložit ladicí kód, musíte vložit devět řádků kódu. Tím bude zdrojový kód vaší aplikace o něco delší, a aplikace se tak stane pomalejší a obtížně udržovatelná. Na tomto místě přichází na řadu základní pravidlo znovupoužitelnosti kódu.

Znovupoužitelnost kódu

Určitě vás napadlo, že byste se měli vyhýbat duplicitnímu kódu, který často vzniká při postupu ve stylu „Copy & Paste“. Při každé změně tohoto kódu (například při změně názvu souboru nebo při přidání třetího ladicího režimu) je nutné provést změnu na mnoha místech kódu.

Zabránit tomu můžete tím, že duplicitní kód přesunete do samostatné metody, kterou na potřebných místech jen zavoláte. Nová metoda třídy `Library` bude vypadat následovně:

```
protected function debug($message) {
    switch(DEBUG_MODE) {
        case 'echo':
            print $message . "\n";
            break;
        case 'log':
            error_log($message . "\n", 3, './library.log');
            break;
    }
}
```

Nové metodě `debug()` je nutné předat řetězec se zprávou, která se má nějak zpracovat. Tato metoda na základě konstanty `DEBUG_MODE` rozhodne, zda se zadaná informace vypíše na obrazovku nebo se zapíše do protokolovacího souboru. Následně je zapotřebí upravit i metody, které mají novou metodu `debug()` využívat.

```
namespace cz\k1886\example;

class Library {
    protected $library = array();
    protected $rentalActions = array();

    public function addToLibrary($id, Publication $p) {
        $this->library[$id] = $p;
        $this->debug(
            'Nová publikace v knihovně: '
            . $p->getCategory()
        );
    }

    public function rentPublication(Publication $p, Member $m) {
        // ... vlastní logika metody
        $this->debug(
            $m->getName() . ' si vypůjčil publikaci: '
            . $p->getCategory()
        );

        return $rentalAction;
    }

    public function returnPublication(Publication $p) {
        // ... vlastní logika metody
        $this->debug(
            $rentalAction->getMember()->getName()
            . ' vrátil publikaci: '
            . $p->getCategory()
        );
        // ... vlastní logika metody
    }
}
```

Díky této úpravě jste zamezili duplikování kódu, třída `Library` je opět skoro tak velká, jako byla na začátku, a změny stačí provést jen jednou nezávisle na tom, na

kolika místech se metoda `debug()` volá. Tím jste se naučili další pravidlo objektově orientovaného návrhu aplikací:

PRAVIDLO

Znovupoužitelnost kódu je lepší než duplicitní kód.

Co se ale stane, pokud bude nutné zapracovat do aplikace nové způsoby ladění? Kromě zápisu do protokolovacího souboru byste mohli chtít odeslání hlášení pomocí e-mailů, SMS nebo zápis do `syslog`. Pokud byste každou z těchto možností zakomponovali do metody `debug()`, stávala by se stále větší a větší, a tím i náchylnější na chyby. Jako se tomu dá zabránit?

Rozměnit na drobné

Metody, které obsahují vlastní aplikační logiku a volají metodu `debug()`, nemusí vědět, jakým způsobem se předávaná informace zpracovává. Co tedy bráni tomu, pokusit se pomocí dědění udělat kód sloužící pro ladění flexibilnější? K dosažení tohoto cíle potřebujete místo jedné třídy `Library` implementovat následující tři třídy:

- `AbstractLibrary` – abstraktní třída, která obsahuje aplikační logiku, ale také abstraktní metodu `debug()`, kterou používají ostatní metody.
- `EchoingLibrary` – třída, která je odvozená od třídy `AbstractLibrary` a implementuje abstraktní metodu `debug()`. Ladicí informace se vypisují pomocí příkazu `print`.
- `LoggingLibrary` – třída, která je odvozená od třídy `AbstractLibrary` a taktéž implementuje abstraktní metodu `debug()`. V této implementaci se ladicí informace místo vypisování na obrazovku zapisují do protokolovacího souboru.

Začněte s implementací abstraktní třídy:

```
namespace cz\k1886\example;

abstract class AbstractLibrary {
    // ... atributy a metody třídy
    abstract protected function debug($message);
}
```

V této třídě dojde jen k malým změnám. Do její definice je nutné zahrnout klíčové slovo `abstract` a toto slovo je též nutné uvést při definici metody `debug()`.

Metoda `debug()` navíc nebude mít žádné tělo, které se bude nacházet až v konkrétní implementaci. Jako další je nutné implementovat obě konkrétní třídy, v nichž přepíšeme metodu `debug()`:

```
namespace cz\k1886\example;

class EchoingLibrary extends AbstractLibrary {
    protected function debug($message) {
        print $message . "\n";
    }
}

class LoggingLibrary extends AbstractLibrary {
    protected function debug($message) {
        error_log($message . "\n", 3, './library.log');
    }
}
```

Obě třídy jsou zaměřené jen na konkrétní úlohy. Třída `EchoingLibrary` se stará výlučně jen o vypsání zprávy na obrazovku a třída `LoggingLibrary` se stará jen o zápis do protokolovacího souboru. Tím jsou oba procesy ladění od sebe oddělené. Nyní musíte samozřejmě trochu upravit i skript s příkladem, protože již není možné vytvořit instanci třídy `Library`. Tu jsme totiž přejmenovali na třídu `AbstractLibrary` a definovali jako abstraktní. Místo toho vytvořte instanci konkrétní třídy v závislosti na konstantě `DEBUG_MODE`:

```
use cz\k1886\example\LoggingLibrary;
use cz\k1886\example\EchoingLibrary;
use cz\k1886\example\Book;
use cz\k1886\example\Member;

switch(DEBUG_MODE) {
    case 'echo':
        $library = new EchoingLibrary();
        break;
    case 'log':
        $library = new LoggingLibrary();
        break;
}

$book = new Book('PC', 100);
$mabo = new Member(1, 'Marian Böhmer');
$luigi = new Member(2, 'Luigi Valentino');
```

```
$library->addToLibrary('pc1', $book);
$library->rentPublication($book, $mabo);
$library->returnPublication($book);
$library->rentPublication($book, $luigi);
```

Pokud jste tento příklad vyzkoušeli, měli byste vidět stejný výsledek jako v předchozím příkladu, tedy za předpokladu, že je konstanta `DEBUG_MODE` nastavená na `echo`. V případě, že chcete ladicí informace posílat přes SMS nebo e-mail, musíte implementovat novou třídu, která v metodě `debug()` poskytuje tuto možnost.

Tím jste se naučili další pravidlo:

PRAVIDLO

Vyvarujte se monolitickým strukturám a rozložte je na co nejmenší části, které mohou být implementované nezávisle na sobě. Pokud používáte rozsáhlé příkazy `if/elseif/else` nebo `switch`, popřemýšlejte, zda by se nedaly nahradit zaměnitelnými třídami.

Po tomto elegantním vyřešení ladění ve třídě `Library` se můžete pustit do dalších tříd, které je také nutné odladit. Řekněme, že chcete například vypsat informace v metodě `markPublicationReturned()` třídy `RentalAction`. Nejdříve vás ale napadne, že jste daný problém přece jen nevyřešili tak elegantně, jak jste si to právě mysleli. Abstraktní metodu `debug()` musíte totiž přidat i do třídy `RentalAction` a potom i do všech konkrétních implementací, které budou jednotlivé způsoby ladění zpracovávat. Rozměnit na drobné tedy znamená, že budete muset znovu vytvořit třídy `AbstractRentalAction`, `EchoingRentalAction` a `LoggingAbstractAction`. Tím ale porušíte jedno z prvních pravidel, které jste definovali, a vlastní kód na ladění budete implementovat vícekrát, jednou ve třídě `Library` a jednou ve třídě `RentalAction`. Pokud k tomu budete ještě chtít ladění pomocí SMS, musíte tuto funkci implementovat znovu dvakrát. Z toho vyplývá, že zatím vyvinutá architektura pro ladění aplikace není úplně perfektní a daný kód je nutné optimalizovat.

Kompozice místo dědění

Jak jste právě viděli, je změna metody ladění velmi omezená, protože každá třída může dědit metody jen od jedné třídy. Na vyřešení tohoto problému byste měli třídu, která implementuje logiku pro ladění, oddělit od třídy, která představuje aplikaci jako takovou (aplikuje business logiku). Oddělte tedy kód ladění od aplikační logiky tím, že kód ladění přesunete do úplně nové třídy. Zachovejte

přitom obě třídy oddělené – tj. jedna třída vypíše údaje přímo, druhá je zapíše do protokolovacího souboru.

```
namespace cz\k1886\example;

class DebuggerEcho {
    public function debug($message) {
        print $message . "\n";
    }
}

class DebuggerLog {
    public function debug($message) {
        error_log($message . "\n", 3, './library.log');
    }
}
```

Obě třídy obsahují metodu `debug()`, postupujete proto podle pravidla, že třídy řeší elementární problémy a mají být co do velikosti co nejmenší. Aby bylo v budoucnosti možné tyto třídy co nejjednodušeji vyměnit, můžete je zavedením rozhraní začlenit do jedné skupiny. Toto nové rozhraní `Debugger` vyžaduje od každé třídy, která chce zpracovávat hlášení o ladění, aby implementovala metodu `debug()`:

```
namespace cz\k1886\example;

interface Debugger {
    public function debug($message);
}
```

Protože tyto dvě třídy už danou metodu obsahují, splňují podmínky dané rozhraním `Debugger`.

```
namespace cz\k1886\example;

class DebuggerEcho implements Debugger {
    public function debug($message) {
        print $message . "\n";
    }
}

class DebuggerLog implements Debugger {
    public function debug($message) {
        error_log($message . "\n", 3, './library.log');
    }
}
```

Místo abstraktní třídy tu máte sice rozhraní, ale nic víc se v kódu nezměnilo. Ve skutečnosti jste jen přesunuli metodu `debug()` do jiné třídy, jejíž instanci vytvoříte velice jednoduše:

```
use cz\k1886\example\DebuggerEcho;
$debugger = new DebuggerEcho();
```

Následně můžete předat metodě `debug()` zprávu:

```
$debugger->debug('Lorem ipsum dolor sit amet');
```

Po provedení tohoto kódu uvidíte text `Lorem ipsum dolor sit amet` vypsaný na obrazovce. Pokud zaměníte řádek, který vytváří instanci třídy, a vytvoříte instanci třídy `DebuggerLog`, zapíše se tato zpráva do protokolovacího souboru. Kód používaný pro ladění aplikací nyní funguje soběstačně, aniž by k tomu potřeboval třídu `Library`. Třídu `Library` je ještě nutné upravit tak, aby používala tento nový způsob ladění.

```
namespace cz\k1886\example;
use cz\k1886\example\RentalAction;

class Library {
    protected $library = array();
    protected $rentalActions = array();
    protected $debugger;

    public function __construct() {
        switch (DEBUG_MODE) {
            case 'echo':
                $this->debugger = new DebuggerEcho();
                break;
            case 'log':
                $this->debugger = new DebuggerLog();
                break;
        }
    }

    protected function debug($message) {
        $this->debugger->debug($message);
    }
    // ... ostatní metody třídy
}
```

V úvodu jste do třídy `Library` vložili nový atribut `$debugger`. Tomu se pak v konstruktoru podle hodnoty konstanty `DEBUG_MODE` přiřadí instance třídy `DebuggerEcho` nebo `DebuggerLog`. Jako poslední věc musíte upravit metodu `debug()` takovým způsobem, aby delegovala úlohu na objekt v atributu `$debugger`. Žádné další změny v kódu nejsou nutné.

Pokud nyní provedete kód z testovacího skriptu, objeví se na obrazovce znovu stejný výpis nebo se provede zápis do protokolovacího souboru, podle toho, na jakou hodnotu je nastavená konstanta `DEBUG_MODE`. Podobným způsobem byste postupovali i při implementaci tohoto kódu ve třídě `RentalAction`.

V tomto okamžiku jste zvládli oddělit kód z původní třídy a zapouzdřit ho do univerzálně použitelné třídy. Tím jste splnili předchozí pravidla, jež přikazují vyvarovat se duplicitnímu kódu. Na základě tohoto řešení můžete odvodit další pravidlo objektově orientovaného návrhu:

PRAVIDLO

Dědění vede k neflexibilním strukturám. Na kombinaci různých funkcí používejte raději kompozice objektů.

Nyní už na aplikaci skutečně není mnoho co vyměnit. Je velmi jednoduché napsat nový ladicí kód, který například odešle informace e-mailem. Avšak jedna možnost vylepšení ještě existuje a bude vysvětlená na následujících řádcích.

Volná vazba místo závislostí

Objekt, který zpracovává hlášení o ladění, se v současnosti vytváří v konstruktoru třídy `Library`, z čehož vyplývá, že tato třída musí znát všechny verze těchto objektů. Pokud budete chtít napsat nový způsob zpracování, například ten na posílání e-mailů, budete muset upravit třídu `Library` a rovněž všechny ostatní, které tyto objekty využívají. Toto na jedné straně znamená množství práce a na druhé straně to je samozřejmě náchylné k chybám.

Mnohem elegantnější by bylo, kdyby knihovna nemusela o vlastních možnostech ladění vůbec nic vědět. Tento způsob programování jde ruku v ruce s úplně základním pravidlem objektově orientovaného programování, které říká:

PRAVIDLO

Vždy programujte vůči rozhraní, a nikdy ne vůči konkrétní implementaci.

Když se vrátíte k deklaraci jednotlivých tříd `DebuggerEcho` a `DebuggerLog`, můžete si uvědomit, že jste již definovali jedno rozhraní, které implementovaly obě z těchto tříd. Zbývá tedy jen upravit třídu `Library` tak, aby znala jen `Debugger` rozhraní, ale nevěděla nic o konkrétních implementacích. Tento princip se nazývá *princip převrácení závislostí* (angl. Dependency Inversion Principle – DIP) a vyžaduje, abyste se místo konkrétní implementace vždy opírali o abstrakci. V tomto okamžiku zatím princip DIP nevyužíváte, protože v mnohých případech jste závislí na konkrétních třídách, jako jsou `DebuggerEcho` a `DebuggerLog`, ale také `RentalAction` nebo `Member`. V dalších kapitolách této knihy se naučíte, jak se lze dost často od těchto závislostí osvobodit.

Při aktuálních problémech těsných závislostí mezi třídou knihovny a jednotlivými konkrétními implementacemi rozhraní `Debugger` můžete aplikovat princip DIP velmi jednoduše – objekt implementující rozhraní `Debugger` nevytvoříte v konstruktoru třídy `Library`, ale konstruktoru této třídy jej prostě předáte:

```
namespace cz\k1886\example;

class Library {
    // ... atributy třídy
    public function __construct(Debugger $debugger) {
        $this->debugger = $debugger;
    }
    // ... další metody třídy
}
```

Pomocí určení typu proměnné v definici konstruktoru je zabezpečené, že tento bude akceptovat jen třídy, jež implementují rozhraní `Debugger`. Tím je zaručené, že v předávaném objektu existuje metoda `debug()`. Knihovně je nyní úplně jedno, jakým způsobem se budou předávaná hlášení zpracovávat. Ví totiž jen to, že předaný objekt `$debugger` nabízí metodu `debug()`, na kterou lze zpracování hlášení delegovat. Samozřejmě je nutné trochu upravit i vytvoření instance knihovny:

```
use cz\k1886\example\DebuggerEcho;
use cz\k1886\example\Library;

$debugger = new DebuggerEcho();
$library = new Library($debugger);
```

Tím jste sestavili další pravidlo a ihned jej aplikovali na svoji aplikaci:

PRAVIDLO

Vyhýbejte se těsným závislostem mezi jednotlivými třídami aplikace a vždy upřednostňujte volné vazby tříd.

V případě ladicího kódu jste toho dosáhli pomocí techniky, která se nazývá *vkládání závislosti* (angl. Dependency Injection – DI). Objekt implementující rozhraní `Debugger` jste vložili do objektu `Library`. Tento objekt nemusí vědět, jakého typu je vkládaný objekt, stačí jen, že implementuje požadované rozhraní. Téma vkládání závislostí je podrobněji rozepsané v příloze na konci knihy.

V této chvíli vás může napadnout otázka, co se stane s aplikací, když budete chtít ladění úplně vypnout. Konstruktor třídy `Library` totiž vždy očekává objekt, jemuž lze předávat zprávy ke zpracování. Na první pohled není možné ladění aplikace kompletně deaktivovat. Naštěstí to jen tak vypadá, jinak by byla celá tato práce zbytečná.

Protože knihovna stále očekává objekt implementující rozhraní `Debugger`, nevyhnete se mu ani v případě, jestliže není požadován žádný způsob ladění. Avšak nikde není definované, že objekt implementující rozhraní `Debugger` musí nějakým způsobem předané zprávy zpracovat. Musí je jen přijmout. Proto můžeme vytvořit takovou implementaci, která všechna předaná hlášení ignoruje:

```
namespace cz\k1886\example;

class DebuggerVoid implements Debugger {
    public function debug($message) {
        // ... všechna hlášení ignorovat
    }
}
```

Tuto variantu můžete použít, pokud nepotřebujete žádný způsob ladění aplikace. Jednoduše předáte třídě `Library` instanci této třídy:

```
use cz\k1886\example\DebuggerVoid;
use cz\k1886\example\Library;

$debugger = new DebuggerVoid();
$library = new Library($debugger);
```

Toto je možné díky tomu, že knihovna nemusí vědět, jakým způsobem se předaná hlášení zpracovávají.

Shrnutí

Na předchozích stranách jste dokázali krok za krokem převést neflexibilní řešení, které bylo založené především na použití „Copy & Paste“, na flexibilní a jednoduše použitelný systém. Ladicí kód nemusí používat jen třída `Library`, ale bez problémů jej mohou využívat i jiné třídy aplikace, přičemž stačí inicializovat jen skutečně použité třídy, čímž výrazně vzroste výkon aplikace.

Použití tříd není omezené jen na příklad knihovny, ale může být přenesené na každou z vašich dalších aplikací. Docílili jste zde maximální znovupoužitelnosti kódu a seznámili jste se s nejdůležitějšími pravidly vývoje softwaru, které můžete přenést i do ostatních částí svých aplikací.

Kromě toho jste zde použili i jeden návrhový vzor. Tento návrhový vzor se nazývá strategie (angl. Strategy) a v této kapitole jste jej použili pro delegování zpracování hlášení na objekt třídy implementující rozhraní `Debugger`.

V následující tabulce najdete seznam návrhových vzorů probíraných v této knize ještě předtím, než se jimi budete blíže zabývat:

Název vzoru	Cíl vzoru
Abstract Factory (Abstraktní továrna)	Vytváří rodiny příbuzných objektů.
Builder (Stavitel)	Odděluje tvorbu komplexních objektů od jejich reprezentace.
Factory Method (Tovární metoda)	Deleguje vytváření objektů na potomky.
Prototype (Prototyp)	Vytváří objekty kopírováním prototypového objektu.
Singleton (Jedináček)	Zabezpečuje, že existuje jen jedna instance určité třídy.
Adapter (Adaptér)	Upraví rozhraní na rozhraní očekávané klientem.
Bridge (Most)	Oddělí rozhraní třídy od její vlastní implementace, při čemž lze obě nezávisle na sobě změnit.
Composite (Strom)	Spojuje více objektů do stromové struktury, kterou lze použít jako jeden objekt.
Decorator (Dekorátor)	Rozšiřuje objekty za běhu programu o novou funkčnost.
Facade (Fasáda)	Nabízí abstraktní rozhraní, které zjednodušuje používání určitého subsystému.
Flyweight (Muší váha)	Umožňuje společné použití malých objektů.

Název vzoru	Cíl vzoru
Proxy (Zástupce)	Kontroluje přístup k objektu pomocí zástupce: - přístup k objektu na jiném serveru (Remote Proxy), - vytvoření objektu až v okamžiku potřeby (Virtual Proxy), - vykonávání administrativních úloh (Secure Proxy).
Chain of Responsibility (Zřetězení zodpovědnosti)	Umožňuje odeslat požadavek řetězu objektů. Zřetězené objekty samy rozhodnou, který z nich jej zpracuje.
Command (Příkaz)	Zapouzdřuje požadavek jako objekt.
Interpreter (Interpret)	Definuje gramatické pravidla a určuje způsob jejich interpretace.
Iterator (Iterátor)	Umožňuje sekvenční přístup k prvkům objektu bez znalosti jeho implementace.
Mediator (Prostředník)	Zajišťuje komunikaci mezi dvěma objekty, které nemusí být v přímé interakci a znát poskytované metody.
Memento (Memento)	Zachytává a uchovává vnitřní stav objektu bez porušení jeho zapouzdření.
Observer (Pozorovatel)	Umožňuje šíření událostí, které nastaly v jednom objektu, na všechny na něm závislé objekty.
State (Stav)	Umožňuje změnit chování objektu při změně jeho vnitřního stavu.
Strategy (Strategie)	Definuje rodinu algoritmů, které jsou navzájem zaměnitelné.
Template method (Šablonová metoda)	Definuje kroky určitého algoritmu a přenechává jejich implementaci svým potomkům.
Visitor (Návštěvník)	Přidává do objektové struktury novou funkčnost a zapouzdří ji do třídy.

ČÁST II

Tvořivé vzory